Exhibit D

Diligence

Exhibit D1

Estream 1.0 Planning Document

8/31/2000

Low-Level Design Status/Plan

Sub Components	Owner	LLD Design Doc completed	LLD review Compited	Estimates for Impl	Impl and Unit Test Completed
Content					
Install Monitor	Sanjay	Done	Done	3 wk	
Builder GUI	Sanjay	Done	Done	1 wk	
FSRFD (Drivers)	Sanjay	Done	Done	2 wk	
AppInstallBlk structure	David	Done	Not needed		
Profiler	David	Done	Done	2 wk	
File Access Monitor	David	Done	Done	1 wk	
Packager	David	Done	Done	1 wk	
eStream distribution	**Bob**	**8/31/2000**	**Status TBD**	**TBD**	
Server Group					
Web Server	Bhaven	Done	Done	8 wk	
Monitor	Mike	Done	Done	4 wk	
SLiM Server	Amit	Done	Done	2 wk	
App Server	Sameer	Done	Done	4 wk	
Admin UI	Bhaven	TBD	TBD	TBD	
End User UI	Bhaven	TBD	TBD	TBD	
Common Server Components	Mike	Done	Done	3 wk	
Messaging	Sameer	Done	Done	3 wk	
Threads Package	Sameer	No Document		1 wk	
Security Design	**Igor/Amit**	**Not Done**	**Not Done**	**TBD**	
Client Group					
Cache Prefetching	Anne	Done	Done	1 wk	9/29/2000
LSM + Plug in	Anne	Done	Done	1 wk	9/22/2000
Client UI	Anne	Done	Done	1 wk	9/22/2000
Client Installer	Anne	Done	Done	1 wk	10/6/2000

Start Client	Anne	Done	Done	1 wk	10/6/2000
Application Install Mgr	Nick	Done	Done	TBD	
Piracy	Nick	Done	Done	TBD	
File Spoofer	Curt	Done	Done	1 wk	
eStream File System	Curt	Done	Done	8 wk	
NoCluster Driver	**Curt**	**Status TBD**	**Starus TBD**	**2 days**	
eStream Cache Manager	Dan	Done	Done	8 wk	
Client Network Interface	Dan	Done	Done	2 wk	

Implementation Plan

Milestones

10/6/2000	ECM (RAM disk cache) and EFSD executes a local "himom" executable
10/6/2000	Photoshop is installed locally and successfully executed from estream sets and appinstallblk produced
10/6/2000	App Server and EMS integrated to copy "himom" executable using a dummy client
10/11/2000	App Server, EMS and CNI integrated to copy "himom" executable from "himom" estream sets
10/13/2000	office is installed locally and successfully executed from estream sets and appinstallblk produced by bl
10/16/2000	App Server, EMS, ENI, ECM and EFSD integrated to run "himom" from estream sets on server
10/20/2000	Following applications built and tested with local installation
	Adobe Premier
	Macromedia Director and Shockwave
	Corel Suite
	Lotus Suite
10/23/2000	Photoshop is installed by AIM and executed from estream sets on App server
	No Subscription
	No License Management
	RAM cache for ECM
	Installation of Photoshop using AIM
10/31/2000	Photoshop is installed by AIM and executed from estream sets on App server
	No Slim Server
	Disk based cache for ECM
	Estream includes initial prefetched pages and these pages are prefetched during installa

Fully functional estream bits (includes initial prefetched pages)
Client software is run as a service
App Server is started by Monitor
Admin UI to stop and start app Server
Application subscription from web server
installation on client after subscription

10/30/2000 Testing environment is setup (configuration of 3 servers and one client)

11/7/2000 Photoshop runs with the following additional functionality

Leads for milestone: Amit and Nick
Slim Server
http protocol
CNI supports uique message ids for NAD
Fully functional LSM
Real Accesstokens
Uninstall applications
Anti-Piracy support
AppServer and SlimServer fail-over
File spoofing

11/9/2000 Clean builds by integration (George) (Raj will drive this)

11/14/2000 Office is running with full functionality

Restruturing of client so it can be started at boot time
Performance tuning
Improve robustness
application upgrade
Crash resillency
All software purified and memory leaks eliminated
(May be) Applets for monitoring server components

11/15/2000 **Office is removed from desktop of at least one person and reinstalled using estream**

11/15/2000 **Code Freeze**

Engineer	9/4 - 9/10	9/11 - 9/17	9/18 - 9/24	9/25 - 10/1	10/2 - 10/8	10/9 - 10/15	10/16-10/22
Server							
Sameer				Messaging			
Mike				common server framework		Monitor	
Bhaven				database schema	services	Java beans/JDBC interfaces	
Amit				Threads		Api Server	
Jae Jung						Vacation	
Chungying Chu							Web Server compo
Builder							
David				Packager			
Bob				Install Monitor/FSRFD	Extractor	Test Builder on C-S apps	
Sanjay							
Client							
Dan					ECM		
Curt					File System Driver		
Anne				Client UI	ILSM + Plugin	Preamble	Client components
Nick					App Install Manager	Test Aim	File spoofer
Raj						CNL	
Ameet						NT/Win9x File System Support	

Exhibit D2

eStream Server Component Framework
Low Level Design

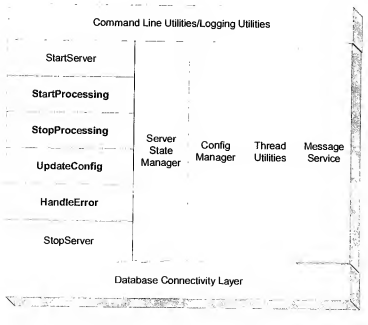
Michael Beckmann
September 4, 2000

Functionality

The *Server Component Framework* provides a common basis on which server components are implemented. The framework provides a number of services such as common server initialization and configuration, messaging, state management, logging, and error handling. The component framework ties together many of the core utilities provided for the server components.

The advantage of the framework is that heterogeneous server components can be managed in a consistent manner with the expectation that all server components will communicate and behave consistently within the system.

All server components with the exception of the web server will be built on top the *Server Component Framework*. To make use of the *Server Component Framework*, a specialized server component will need to extend the framework by implementing the methods high-lighted in gray. Implementing these interfaces makes the specialized server component "plug-able" within the framework.



The following table give a brief description of each of the routines that need to be specialized by each server component to make it plug-able into the Server Framework:

StartProcessing	Specialized server component routine to request the server component to start processing work.
StopProcessing	Specialized routine to request the server component stop processing work and transition into an idle state
UpdateConfig	Specialized routine to dynamically update configurations while a component is either in the processing or idle state.
HandleError	Specialized routine to handle the occurrence of an error

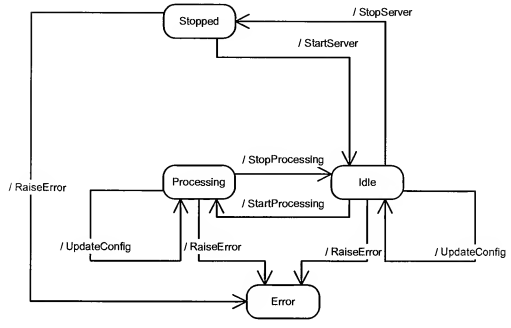
Server State Manager:

At the heart of the server component framework is the *Server State Manager*. The server state manager is a set of interfaces that initiate and manage state changes within a server component. All Server components, by virtue of being built on top of the component framework, can be managed uniformly across a deployment.

The *Server State Manager* implements a simple state machine that is shared between components. It manages the state transitions within the server component. Additionally, the state manager maintains current state information for each server component and logs state transition history in the event that a server component terminates unexpectedly.

As specified above, each server component is required to implement a number of transition methods, with pre-defined signatures, which the state manager will execute when making a state transition.

The following diagram shows the state diagram and the associated transitions:



Message Service:

The *Server Component Framework* depends on a message service which is used by the Server State Manager and Configuration Manager to communicate with the System Monitor.

The *Server State Manager* uses the messaging service to listen for state change requests from the System Monitor which it satisfies by returning the current state, any up-to-date status, and load information.

The *Configuration Manager* uses the message service to request configuration information from the *System Monitor*. Although each server component could easily go to the database for configuration information, it has been decided to go through the monitor as to save db licensing costs.

See below for more details on messaging protocols for the *Server State Manager* and the *Configuration Manager*. Also, refer to the low-level design document for details on the design of the eStream Messaging Service (EMS).

Configuration Management:

The configuration management utility is used by all server components to manage the server configurations. It provides the following functionality:

- Configuration for a server consists of a set of name – value tuples where the values themselves can be a set of name-value tuple.
- Servers can load the complete configuration from the database (indirectly).

- Servers can load the configuration for a given name.
- Servers can load the configuration from a flat file also.

On the Server Manger interface, configuration will appear as a table containing name – value tuples. The table may be hierarchical to represent nested structures containing the values which can themselves be name values. An example of a simple name-value pair would be:

port 8080

An example of nested name values would be:

Applications:
word.exe *windows2000sp3*
excel.exe *win98sp4*

On a flat file the configurations will always be name-value pairs. To represent one level nested structure the format would be:

Applications word.exe windows2000sp3
Applications excel.exe win98sp4

A common set of configurable parameters is defined for all server components. These configurations are maintained by the *Server Component Framework* in collaboration with the *Configuration Manager*. All configuration information is persistently stored within the database. The common configurations are used to initialize the server component after the component process has been launched. Refer to the configuration table below for more details on common configurations. Specialized server components can support additional configurations (non-common) depending on the server type. These configurations are read from the database and updated when a server component starts processing. They can also be updated dynamically while a server component is processing through the use of the **UpdateConfig** interface.

The list of common configurations include:

Information	Supports Dynamic Config	State	Description
ServerID	No		Unique identifier for server components. This server identify is unique within a deployment. This ServerID is not known to eStream clients. Its purpose is as a handle to uniquely identify server components.
ServerType	No		Identifies the type of server component. One of the following applies: ▪ Primary Monitor ▪ Backup Monitor

			▪ Application Server ▪ SLiM Server
DbUser	No		User name string required for database connectivity for this server ID
DbPasswd	No		Database password associated with the DbUser
Dsn	No		Data Source Name used to access the database.
PortNum	No		PortNumber used for light-weight messaging listener
MachineID	No		Machine ID is used to get at important machine information needed for all server components such as: ▪ IP address for the machine server component is hosted on ▪ Domain name for the machine ▪ Machines name
AutoReStart	Yes	Any State	Flag indicating that server component process can be restarted automatically without manual intervention
TimeOut	Yes	Any State	Specifies the timeout period for the listener. If the timeout period is reached. The component assumes that it has lost the connection. All Server components have a listener by which they receive instructions from the primary system monitor. Even the monitor has a listener that communicates with the Server Admin UI.

Command Line Utilities:

The *Command Line Utilities* component provides a consistent way to define and process command line arguments. To use this utility, the using component must define a table of arguments, which defines the valid set of arguments, whether or not they are required, and any default values.

Arguments are specified on the command line as name/value pairs. The utility implements the following command line syntax to support the name/value pairs. The argument syntax is defined as follows:

<name>=<value>

name	Name is an alpha-numeric identifier. The Name can be of arbitrary length as supported by the system however shorter names are recommended. Names are case sensitive
value	Any alpha-numeric value. Punctuation characters may also be used. Values are case sensitive

There can be no spaces between the <name>, "=", and the <value> elements. The existence of one or more spaces or tabs delineates separation between arguments on the command line.

Example: server.exe sid=267 dns=oracle user=michaelb passwd=mypasswd

- If a named argument is specified more than once on the command line, subsequent arguments will cause a diagnostic to be issued and the argument will be ignored.
- This utility allows the user to specify default values for arguments. If a default value is defined then the argument will be processed with its default in the event that the argument is not specified on the command line.
- This utility allows the user to tag specific arguments as required. If the required argument is not specified on the command line this utility will raise a diagnostic for the required argument. Not specifying a required argument will cause a fatal error.

The following options are supported:

sid	Server Component Identifier. Each server component within a deployment is uniquely identified via the sid. The sid is a handle into the database for accessing information unique to a specific server component.
dsn	Data Source Name. A data source name is necessary to establish an ODBC connection. Data Source Names are generated by an ODBC administrative tool
dbuser	User name. For database access security, all components need to connect as a specific user.
dbpasswd	password associate with the dbuser

Logging Utilities:

All servers and clients in eStream 1.0 need to log the error and access data. Logging enables component debugging and auditing support.

EStream Framework should provide logging with the following features:

- Each component will have an error and optioanally an access log file. The names of these files would be <component>_error.log and <component>_access.log.
- The files will be located in the <eStream1.0 Root Dir>\logs directory.
- The error log files will have messages with the following priorities:
  - 4-Low : A warning which can be ignored.
  - 3-Medium: A warning which needs to be looked into.
  - 2-High: Recoverable Error in the component.
  - 1-Critical: Fatal Error. Needs admin assistance.
- Logging level should be configurable. The following levels are to be supported.
  - 0: Only errors will be logged. This will be the default level.
  - 1: Errors and Warnings to be logged.
  - 2: Errors, Warnings and Debugging information to be logged.
  - 3: Errors, Warnings and advanced Debugging (like memory dumps, tcp stack dumps etc) to be logged.

- Log Wrapping to be supported. The log files will wrap at a predefined size. On wrapping the following actions will occur:
  - Any existing <logfile>.bak will be deleted from the system.
  - The current <logfile> will be backed to <logfile>.bak
  - The component will continue logging to the <logfile>.

For each eStream client and server component logging the log files (component_error.log and component_access.log) should be written in eStream1.0Root\logs directory. The formats for the log files will be as follows:

Error Log:

[HEADER]
[TimeStamp] [Thread ID] [Priority] [Message]
...
[FOOTER]

An example of this log format would be:

```
**********************************************
Omnishift eStream Application Server
Server Started.
StartTime: 14/Aug/2000:16:31:19 -0700
IP Address: 1.1.1.1
Logging Level: 3
**********************************************

[14/Aug/2000:16:31:19 -0700] 0 2-High Cannot connect to the database.
Invalid Username/Password.
[14/Aug/2000:16:31:19 -0700] 1 1-Critical Cannot start the HTTP listener
at port 80.
[14/Aug/2000:16:31:19 -0700] 0 1-Critical Shutting down the server.

**********************************************
Omnishift eStream Application Server
Server Stopped.
StopTime: 14/Aug/2000:16:35:19 -0700
IP Address: 1.1.1.1
Logging Level: 3
**********************************************
```

Format of Access Log Message:

[HEADER]
[TimeStamp] [Thread ID] [Message]

[FOOTER]

Data type definitions

Server State:

The server components can be in any one of the following states:

State	Description
STOPPED	If a server is in the STOPPED state then the component process is not running.
IDLE	Server component is up and running. The server has been initialized with the common configuration and the messaging system has been enabled. The listener is actively waiting on the System Monitor for transition requests. The server component is not processing any work specific to this servers specialization.
PROCESSING	Server component is actively taking requests and processing work specific to its specialization. ie. serving access tokens, and application file requests.
ERROR	An error has occurred in the system. Unless the server component is configured with **AutoReStart** and ERROR state must be manually cleared by the server-side administrator.

Server State Transitions:

Changes in server component state are initiated either by the *System Monitor* or directly by the server-side administrator for the system monitor. The exception to this is when an error condition is raised by a server component. In this case, the component will initiate the state change itself. The following state transitions are supported:

Action	Description
START_SERVER	Server is expected to be in the STOPPED state. If a server component is configured to support AutoReStart then the ERROR state is also a valid state from which to initiate this action.
STOP_SERVER	Causes the server to exit its process. The server can be stopped from any state.
START_PROCESSING	Causes the server to change from the IDLE state to the PROCESSING state.
STOP_PROCESSING	Causes the server to change from processing to IDLE state.
UPDATE_CONFIG	Request that the server read its configuration from the configuration manager and change its configuration.

RAISE_ERROR	Request that the server go to ERROR state. This causes an error handler to be called. If the error is fatal it will cause immediate termination of the server process.

Finite State Table:

```
FSMTableEntry ServerStateMgr::FSMTable[] =
{
        { START,{{START_SERVER,STOPPED, START_SERVER,NULL},
                {START_PROCESSING, STOPPED ,START_PROCESSING, NULL},
                {NULL_REQUEST,  NULL_STATE, NULL_REQUEST, NULL}} },

        {STOPPED, {{START_SERVER, IDLE, NULL_REQUEST, &StartServer},
                {START_PROCESSING, IDLE, START_PROCESSING, &StartServer},
                {RAISE_ERROR, ERROR, NULL_REQUEST, &HandleError},
                {NULL_REQUEST,  NULL_STATE, NULL_REQUEST, NULL}} },

        {IDLE,{{START_PROCESSING, PROCESSING,  NULL_REQUEST,
&StartProcessing},
                {STOP_SERVER, STOPPED, NULL_REQUEST, &StopServer},
                {RAISE_ERROR, ERROR, NULL_REQUEST, &HandleError},
                {UPDATE_CONFIG,  IDLE, NULL_REQUEST, &UpdateConfig},
                {NULL_REQUEST,  NULL_STATE, NULL_REQUEST, NULL}} },

        {PROCESSING,{{STOP_PROCESSING, IDLE, NULL_REQUEST,
&StopProcessing},
                {UPDATE_CONFIG,PROCESSING, NULL_REQUEST,
&UpdateConfig},
                {STOP_SERVER, IDLE, STOP_SERVER,  &StopProcessing},
                {RAISE_ERROR, ERROR, NULL_REQUEST, &HandleError},
                {NULL_REQUEST,  NULL_STATE, NULL_REQUEST, NULL}} },

        { ERROR,{{STOP_SERVER, STOPPED,NULL_REQUEST, NULL},
                {NULL_REQUEST,  NULL_STATE, NULL_REQUEST, NULL}} }

        {NULL_STATE,{{NULL_REQUEST, NULL_STATE, NULL_REQUEST,
        NULL} } }
};
```

Messaging Service Protocol:

A light-weight messaging protocol is needed to facilitate communication between server components. The primary purpose of the messaging protocol is to communicate transition requests to the server components. In response, server components communicate state, status, and load information back to the *System Monitor*.

The messaging protocol supports two primary message types. 1) Requests for the *System Monitor* to perform on other servers. 2) Requests to the server components themselves. These message types are distinguished through the protocol as described below. If the receiver ID and the target ID are identical then the request is for the receiver. If the target is different than the receiver, the message is for the *System Monitor* to enact a request on the target component.

All requests are required to be acknowledged. Without an acknowledgement the message is considered un-received.

OpCode	senderID	receiverID	targetID	Data

The following table describes the protocol used by the Server State Manager in its communication with the System Monitor.

OpCode	Description	Data
0x01	Request for current state	None
0x02	Acknowledgment	▪ Current state ▪ Load info ▪ Status info
0x03	Stop Server request. Acknowledged with 0x02 message	None
0x04	Start Server request. Only valid for System Monitor. Acknowledged with 0x02	None
0x05	Start Processing Request. Acknowledged with 0x02	None
0x06	Stop Processing Request. Acknowledged with 0x02	None
0x07	Update Configuration Request. This is a request for a server component to request its specialized configuration information from the System Monitor and update itself. Acknowledged with 0x02.	None

The messaging protocol used by the configuration manager is described below:

OpCode	Description	Data
0x01	Request a complete reload of Configuration	
0x02	Request a reload of specific configuration items.	▪ Number of items being requested. ▪ array of names of configuration items.
0x03	Acknowledgement of configuration reload request.	▪ Number of tuples being returned ▪ Flat representation of configuration tu-

		ples

Interface definitions

Server State Manager:

```
class ServerStateMgr
{
private:
        ServerState CurrentState;
        static FSMTableEntry FSMTable[];

public:
        ServerStateMgr(void);
        ~ServerStateMgr(void);

        ServerState SetState(ServerState);
        ServerState GetState(void);
        ServerState ProcessRequest(ServerRequest);
};
```

SetState	**Description:** Sets the current state of the server component. 1. Log the state change request 2. Update the state field within the server component in memory data structures. 3. Send message to requester informing them of the successful state change. Note: **SetState** does not update the database directly as in the orginal design. The database is updated by the *System Monitor* once it has received an acknowledgement. A state transition is not complete until **SetState** returns successfully and the Monitor has update the database. **Input:** state value to set current state to. **Output:** current state after the new value has been set. If an error occurs will go to error state. **Errors:** 1. Invalid state argument 2. Failure to either connect or commit state change to the database.

GetState	**Description:** returns the current state. This function does not read from the database to get the current state. The assumption is that if the server component is up and running and that it maintains a valid state. **Input:** none. **Output:** returns the current state. **Errors:** None. Will always return a valid state.

ProcessRequest	**Description:** request to the Server State Manager to change server state. This routine implements the guts of the state machine.

	1. Get the current state, and transition request 2. Index into the FSM table and continue to transition from state to state until the transition request is satisfied. 3. Each state transition calls the specialized transition routines for each component. 4. Call to **SetState** to complete each state transition. 5. In the case of an error the state machine will process a RAISE_ERROR request which will call the specialized **HandleError** and transition to the ERROR state. **Input:** server transition request. Refer to table of valid requests defined above. **Output:** current state after the request has been completed. **Errors:**

Server Component Framework:

```
class ServerComponent: ServerStateMgr{    // abstract base class
private:
        ErrorInfo*      Error; // maintains error if error was detected
        ServerConfig*   Config;  // holds common configuration
        Connection*     Listener; // messaging          utility
public:
        virtual int StartServer(void); // may be specialized by a server component
        virtual int StopServer(void); // may be specialized
        virtual int StartProcessing(void) = 0; // must be specialized
        virtual int StopProcessing(void) = 0; // must be specialized
        virtual int UpdateConfig(void) = 0; // must be specialized
        virtual int HandleError(void) = 0; // must be specialized
        void Run(Request);
}
```

StartServer	**Description:** Called by the *Server State Manager* when a server component is to be started. The **StartServer** routine is provided as part of the SeverComponent class. It performs the following: 1. Send request to System Monitor to request an update of common configuration information. 2. Apply the configuration information to the server component. 3. Construct a listener connection object and start the message service. 4. Return success or failure. Note: ▪ This routine must return immediately to the main thread. Otherwise the *Server State Manager* will be blocked. ▪ Successful return from the **StartServer** routine will put the server into the IDLE state. **Input:** None. **Output:** Value of 0 if successful else error condition **Errors:** May return negative error condition

StopServer	**Description:** Called by the *Server State Manager*. 1. Perform any necessary cleanup. 2. Send last acknowledgment confirming shutdown to requester 3. Shut down the messaging system and the listener. 4. exit process Note: The monitor will update the database and perform logging. **Input:** None. **Output:** Value of 0 if successful else error condition **Errors:** May return negative error value

StartProcessing	**Description:** Called by the *Server State Manager*. This routine must be defined by each specialized server component. This routine is used to provide all functionality unique to different types of servers. 1. Spawn a primary processing thread (also known as the boss thread). a. Read server specific configurations unique to this type of server component from the System Monitor b. Spawn worker threads. Depending on the server type this routine does the heavy lifting to either process access tokens and renewals in the case of SLiM server, or process file requests for application servers, or manage and monitor the server components in the case of the *System Monitor*. Note: ▪ This routine must return immediately so that the *Server State Manager* can continue to operate in the main thread. ▪ This routine may make use of the *Server Configuration Manager* for obtaining specialized configuration information **Input:** None **Output:** Value of 0 if successful else error condition. **Errors:** TBD

StopProcessing	**Description:** Called by the *Server State Manager*. This routine must be defined by the specialized server component type. 1. Reverse all actions performed by the **StartProcessing** routine. All worker threads should be joined or pooled in waiting state. Successful return from this routine will put the server component into the IDLE state. **Input:** None. **Output:** Value of 0 if successful else error condition. **Errors:** TBD

UpdateConfig	**Description:** Called by the *Server State Manager*. This routine must be defined by the specific server component type. The purpose of this routine is apply dynamic configurations or update specialized configurations that are unique to this server component. <may require adding a new state to separate dynamic and static configurations> **Input:** None. **Output:** Value of 0 if successful else error condition. **Errors:** TBD

HandleError	**Description:** Component defined error handling routine to handle errors such as timeouts, etc. This routine will need to handle a number of error cases as are possible by the specialized component. The error information is maintained with

	the ServerComponent class. **Input:** None. **Output:** Integer value designating a handled error or failure. If the error cannot be handled then it is fatal. **Errors:** TBD

Run	**Description:** This routine implements the main processing loop for the server component and runs in the main thread. This routine drives the server component by initiating state requests from the *System Monitor*. Note: The *Server State Manager* always runs in the main thread. 1. Call **ProcessRequest** to transition the server component into the initially requested state. 2. Enter main processing loop a. Check for requests from the message service. b. Call **ProcessRequest** to service the request. c. Send acknowledgement for the request to the message service. Acknowledgement includes new state, load info, and status. **Input:** Initial Transition Request **Output:** None. This routine should never return **Errors:** None.

Server Component Main Loop:

The following main loop is common to all server components:

```
void ServerComponent::Run(ServerRequest Request)
{
        ProcessRequest(Request);
        while (1)
        {
                Request = Listener->GetRequest();
                ProcessRequest(Request);
                Listener->AckRequest(Request, GetState, GetLoad, GetStatus);

        }
}
```

```
#include "ServerArgs.h"
#include "Server.h"

int main(int argc, char* argv[]) {
        Args = new ArgList();
        Args->ProcessArgList(argv, argc);
        Server = new ServerComponent(GetValue(SID),
```

```
                                        GetValue(DNS),
                                        GetValue(DBUSER),
                                        GetValue(PASSWD));
        Server->Run(START_PROCESSING);
}
```

Command Line Utilities:

```
class NameValuePair
{
        private:
                char* Name;
                char* Value;
        public:
                NameValuePair();
                ~NameValuePair();
                char* GetValue(void);
                char* GetName(void);
                char* SetName(char*);
                char*SetValue(char*);
};
```

```
typedef int (*pFunc)(NameValuePair*);

struct ArgTblEntry
{
        char*   Name;
        bool    Required;
        char*   DefaultValue;
        pFunc   ProcessFunction;
};
```

```
ArgTblEntry const ServerArgsTbl[] = {
        {"sid",         true,   0,      &ProcessSId},
        {"dsn",         true,   0,      &ProcessDsn},
        {"dbuser",      true,   0,      &ProcessDbUser},
        {"dbpasswd",    true,   0,      &ProcessDbPasswd},
        {0,             0,      0,      0}
};
```

```
typedef vector<NameValuePair*>    ArgVector;

class ArgList
{
        private:
```

```
        ArgVector                   ArgVec;
        const ArgTblEntry*          ArgTbl;

    private:
        NameValuePair*              ParseArg(char* Arg);
        char*                       ParseName(char* Arg);
        char*                       ParseValue(char* Arg);
        int                         ProcessArg(NameValuePair*);
        int                         FinalizeArgs(void);

    public:
                                    ArgList(const ArgTblEntry*);
        int                         ProcessArgList(char* argv[], int argc);
};
```

ProcessArgList	**Description:** Process the entire argument list. In a loop for each argument argv[] … 1. Call **ParseArg** passing in argv[]. 2. **ParseArg** passes the result to **ProcessArg** 3. After processing the entire argument list and exiting the loop call **FinalizeArgs** **Input:** argv and argc as passed into main() entry point **Output:** integer value designating success or failure **Error:**
ParseArg	**Description:** Takes a char* argument and verifies that it follows that name/value syntax defined as <name>=<value> **Input:** Next char* argument on the list **Output: NameValuePair**. NULL will be returned in the event of a syntax error **Error:**
ProcessArg	**Description:** This routine performs the semantic analysis of an argument. 1. Look up name in the **ArgTbl** 2. Verify that the value is valid 3. Add the name value pair to a list of processed arguments called **ArgVec** list. 4. If this name value pair already exists in the list then issue a diagnostic. 5. Call the supplied processing function for this argument as specified in the **ArgTbl** **Input: NameValuePair** **Output:** Interger value designating success or failure (0 for success, positive integer for other errors) **Error:**
ParseName	**Description:** Verify that the Name part of the argument conforms to being alpha-numeric **Input:** char* Name part of argument **Output:** char* Name else NULL **Error:** None
ParseValue	**Description:** Verify that the Value part of the argument conforms to being alpha-numeric and/or punctuation characters **Input:** char* Value part of argument **Output:** char* Value else NULL **Error:** None
FinalizeArgs	**Description:** Post process the argument list. The purpose of this routine is to validate that all required arguments have been defined on the command line. Also processes and adds default arguments to the **ArgVec**. **Input:** None **Output:** Success or Failure **Error:**

Configuration Manager:

```
class  Tuple {
        string name;
        Value value;
};

class Value {
        int type;
};

class StringValue: public Value{
        string value;
};

class TupleValue: public Value {
        vector <tuple> tupleArray;
};

typedef vector < tuple > ConfigArray;

class ServerConfig {
private:
        ConfigArray Array;
public:
        ServerConfig(serverId, dns, dbuser, dbpasswd); // Initialize from db
        ServerConfig(serverId, string filename); // To initialize from a file.

        ConfigArray* GetConfigArray(int serverId);
        Tuple* FindConfig(string Name);
        int Reload(void);
        Tuple* GetConfig(int serverId ,string Name);
};
```

ServerConfig	**Description:** Constructor for Configuration Manager. 1. Initializes configuration manager. 2. Opens the database and gets configuration array **Input:** Server Id, Data Source Name, Database User name, and database users password. **Output:** None **Errors:**
ServerConfig	**Description:** Constructor for Configuration Manager. 1. Initializes Configuration Manager. 2. Opens configuration file and reads configuration array. **Input:** filename of flat-file configuration. **Output:** None **Errors:**

GetConfigArray	**Description:** Returns the entire configuration for a given server id. This routine always retrieves its information either from the flat file or the database. **Input:** ServerId specifying which server to retrieve configuration for **Output:** Returns a vector holding the configuration or NULL **Errors:**

GetConfig	**Description:** Returns the configuration for the specified name. This routine always retrieves its information either from the flat file or the database. **Input:** ServerId specifying the server to retrieve configuration for and Name of configuration item. **Output:** Configuration Tuple. A Tuple may be a nested Tuple. NULL if an error is encountered. **Errors:**

FindConfig	**Description:** Returns the Tuple specified by the name. This routine does not go to the database or flat-file to get its value. Rather it finds the value in the ConfigArray maintained by the Configuration Manager. **Input:** Name of the configuration item. **Output:** Configuration Tuple. NULL if an error is encountered or the Tuple does not exist in the current configuration. **Errors:**

Reload	**Description:** Reloads the entire configuration from the database or flat-file. This routine may reload its configuration indirectly through the use of the System Monitor. In this case it will make a message request to the monitor and listen for the configuration results. **Input:** None **Output:** integer specifying success or failure. Zero will be returned in the case of Success. A negative value in case of error. **Errors:**

Logging Utilities:

```
class LogManager
{
private:
        char* FileName;
        int MaxFileSize;
        char* ResourceFile; // message catalog file

        char* GetMessage(MsgNum, MsgStr)
public:
        LogManager(ServerId,Size=10);
        LogMessage(MsgStr);
        LogMessage(ThreadId, MsgNum, MsgStr, …);
```

```
};
```

LogMessage	**Description:** Write message out to log file. There are two forms of LogMessage. The first will write out a message buffer as is (unformatted) bypassing the resource file. The second form will format the message. Both forms of LogMessage always pre-append a time stamp. 1. Lookup message number in the resource file and get message string 2. format the log message using time stamp, thread id, etc. 3. write out message into the log file. **Input:** Thread Id, Message Number, Message String, and variable number of arguments. **Output:** None. **Error:**

GetMessage	**Description:** Routine returns a message string from the resource file for the message number specified. **Input:** Message number, C Locale text string. **Output:** Message string. Either way, Get Message will always pass a return a valid message string by either returning the string from the resource file or by passing back the MsgStr passed in. **Error:** If an error occurs trying to get a message from the resource file, a message will be logged to the error log.

```
class ErrorLog: protected LogManager
{
private:
        LogLevel ErrorLogLevel;
public:
        ErrorLog(ServerId, LogLevel=0, Size=10);
        LogError(ThreadId, ErrorNum, ErrorMsgStr, ...);
};
```

LogError	**Description:** Writes output to error log file. 1. Check that the message level against the current ErrorLogLevel. 2. Format the message and call the long form of **LogMessage** to write the buffer out to the file. **Input:** 1. ThreadId: Thread identifier to help with the debugging process. 2. ErrorNum: Error number used to uniquely identify an error message in the resource file. 3. ErrorMsgStr: Message string which includes stdio like string formatting. 4. …: variable list of arguments to be inserted into the message string per the format. **Output:** None. **Error:**

Testing design

Each of the components that make up the Server Component Framework will be able to be tested independently of the other components. Each component will have a main entry point defined within a testing .exe to accomplish the Unit testing phase.

Testing of the component framework will be done in phases. Each of the phases is described below along with its dependencies.

Phase 1: Unit testing Test basic components that make up the framework. Each components functionality, restrictions, and boundary conditions will be tested. Will allow testing common configurations for a single server component. This round of unit testing will test the integrated component utilities and framework.	**Dependencies:** 1. ServerComponent class 2. ServerStateMgr class 3. ArgList class 4. Logging Utilities 5. Configuration Manager (flat-file)
Phase 2: Unit testing (full functionality) Test full functionality including messaging interfaces and database connectivity.	**Dependencies:** 1. Phase 1 2. Database connectivity 3. Messaging Service
Phase 3: Integration Testing	**Dependencies:** 1. Phase 2 2. System Monitor (including backup) 3. SLiM Server, App Server, Web-Server
Phase 4: Stress Testing See section on stress testing for details	**Dependencies:** 1. Phase 3

Unit testing plans

Command Line Utilities

The Command line utilities will be tested in a stand-alone module called cmdline.exe. It will support the command line arguments defined in this document.

Configuration Manager

The configuration module is a stand-alone module which will be tested using a config-test.exe executable. The executable will exercise all of the interfaces described above. The configtest.exe executable should be testable in the DB and the non-DB mode.

Logging Utilities

The logging utility will be built as a DLL (otlog.dll). We will provide a binary otlog-test.exe which will exercise each of the interfaces mentioned above.

Server State Manager

The Server State Manager and the Server Component Framework will be tested independently of specialized components. The routines that require specialization (**StartProcessing**, **StopProcessing**, **HandleError** and **UpdateConfig**) will be provided to simply return successfully.

Stress testing plans

Stress testing will require having at least the System Monitor functionality implemented since it is used to drive the server components.

1. Test to repeatedly start, stop, reconfigure the server component.
2. Test to crash machines with server components to validate:
    a. data persistence.
    b. detection capabilities and response.
    c. auto restart.
3. Test to kill individual server component processes.
    a. data persistence.
    b. detection capabilities and response.
    c. auto restart.
4. Test lost database connectivity
5. Test lost of messaging capabilities
    a. repeatedly losing and re-establishing messaging connectivity
6. Test error recovery under adverse conditions.
7. Test recovery from running out of memory, thread resources.
8. Test recovery from threads dying.
9. etc.

Coverage testing plans

1. Goal: 100% path flow coverage. Only exceptions for known error conditions that cannot be practically reached (e.g. thread synchronization, etc.)

Cross-component testing plans

The following pair-wise testing will be performed:

1. framework/database (phase 2)
2. framework/messaging (phase 2)
3. framework (System Monitor) /framework (backup Monitor) (phase 3)
4. framework/Web Server (phase 3)
5. framework (System Monitor) /framework (Other Servers) (phase 3)

Upgrading/Supportability/Deployment design

1. Each error condition will be documented with explanations and practical work-arounds
2. Component framework will support enhanced debug option to dump additional debugging information to special log files.

Open Issues

Exhibit D3

eStream Set Format Low Level Design

Sanjay Pujare and David Lin
Version 0.1
5 September 2000

Functionality

The eStream Set is a data set associated with an application suitable for streaming over the network. The eStream Set is generated by the eStream Builder program. This program converts locally installable applications into the eStream Set. This document describes the format of the eStream Set.

Data type definitions

The format of the eStream Set consists of 4 sections: header, Root Version Table (RVT), Size Offset File Table (SOFT), and Concatenation Application File (CAF) sections. Fields greater than a single byte is stored in little-endian format.

1. Header section

- **MagicNumber [4 bytes]:** Magic number identifying the file content with the eStream Set
- **ESSVersion [4 bytes]:** Version number of the eStream Set format.
- **AppID [16 bytes]:** A unique application ID for this application. This field must match the AppID located in the AppInstallBlock. Guidgen is used to create this identifier.

- **RVToffset [4 bytes]:** Byte offset into the start of the RVT section.
- **RVTsize [4 bytes]:** Byte size of the RVT section.
- **SOFToffset [4 bytes]:** Byte offset into the start of the SOFT section.
- **SOFTsize [4 bytes]:** Byte size of the SOFT section.
- **CAFoffset [4 bytes]:** Byte offset into the start of the CAF section.
- **CAFsize [4 bytes]:** Byte size of the CAF section.

- **VendorNameLength [2 bytes]:** Length of the vendor name.
- **VendorName [X bytes]:** Name of the software vendor who created this application. I.e. "Microsoft".
- **AppBaseNameLength [2 bytes]:** Length of the application base name.
- **AppBaseName [X bytes]:** Base name of the application. I.e. "Word 2000".
- **MessageLength [2 bytes]:** Length of the message text.
- **Message [X bytes]:** Message text.

2. Root Version Table (RVT) section

The Root version entries are ordered in a decreasing value according to their file numbers. The Builder generates unique file numbers within each eStream Set in a monotonically increasing value. So larger root file number implies later versions of the same application. The latest root version is located at the top of the section to allow the eStream Server easy access to the data associated with the latest root version.

- **NumberEntries [4 bytes]:** Number of patch versions contained in this eStream Set. The number indicates the number of entries in the Root Version Table (RVT).

Root Version structure: (variable number of entries)

- **VersionNumber [4 bytes]:** Version number of the root directory.
- **FileNumber [4 bytes]:** File number of the root directory.
- **VersionName [32 bytes]:** Application version name. I.e. "SP 1".
- **Metadata [28 bytes]:** See eStream FS Directory for format of the metadata.

3. Size Offset File Table (SOFT) section

The SOFT table contains information to locate specific files in the CAF section. The entries are ordered according to the file number starting from 0 to NumberFiles-1.

- **NumberFiles [4 bytes]:** Number of entries in this section.

SOFT entry structure: (variable number of entries)

- **Offset [4 bytes]:** Byte offset into CAF of the start of this file.
- **Size [4 bytes]:** Byte size of this file. The file is located from address Offset to Offset+Size.

4. Concatenation Application File (CAF) section

CAF is a concatenation of all file or directory data into a single data structure. Each piece of data can be a regular file, an AppInstallBlock, an eStream FS directory file, or an icon file.

a. *Regular Files*

- **FileData [X bytes]:** Content of a regular file

b. AppInstallBlock (See AppInstallBlock document for detail format)

A simplified description of the AppInstallBlock is listed here. For exact detail of the individual fields in the AppInstallBlock, please see AppInstallBlock Low-Level Design document.

- **Header section [X bytes]:** Header for AppInstallBlock containing information to identify this AppInstallBlock.
- **Files section [X bytes]:** Section containing file to be copied or spoofed.
- **AddVariable section [X bytes]:** Section containing system variables to be added.
- **RemoveVariable section [X bytes]:** Section containing system variables to be removed.
- **Prefetch section [X bytes]:** Section containing pointers to files to be prefetched to the client.
- **Profile section [X bytes]:** Section containing profile data. (not used in eStream 1.0)
- **Comment section [X bytes]:** Section containing comments about AppInstallBlock.
- **Code section [X bytes]:** Section containing application-specific code needed to prepare local machine for streaming this application
- **LicenseAgreement section [X bytes]:** Section containing licensing agreement message.

c. EStream FS Directory

An eStream FS Directory contains information about the subdirectories and files located within this directory. The information includes file number, names, and metadata associated with the files.

- **MagicNumber [4 bytes]:** Magic number for eStream FS directory file.
- **FileNumber [4 bytes]:** File number for each file in this directory.
- **HashValue [2 bytes]:** Hash value for quick filename value comparison to bypass the more expensive comparison of string names.
- **FileNamePointer [4 bytes]:** Pointer to offset in this file where the long file name is located.
- **Metadata [32 bytes]:** The metadata consists of file byte size (4 bytes), file creation time (8 bytes), file modified time (8 bytes), file size (4 bytes), file flags (4 bytes), eStream flags (4 bytes). The bits of the file flags have the following meaning:
  - **Bit 0:** Read-only – Set if file is read-only
  - **Bit 1:** Hidden – Set if file is hidden from user

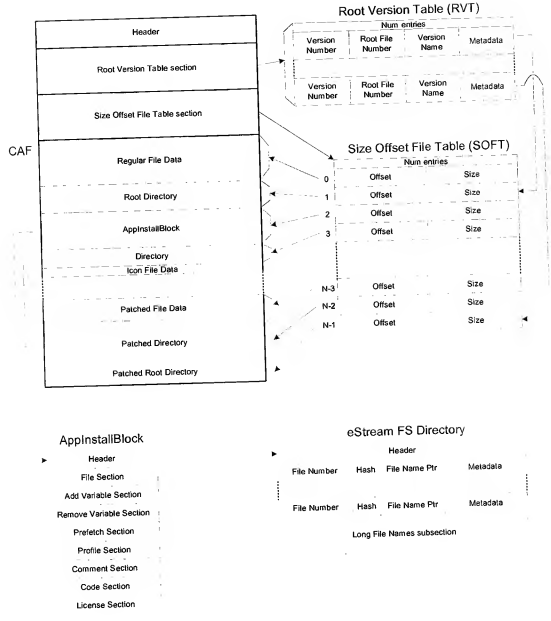
  The bits of the eStream flags have the following meaning:
  - **Bit 0:** ForceUpgrade – Used only on root file. Set if client is forced to upgrade to this particular version if the current root version on the client is older.

    - **Bit 1:** RequireAccessToken – Set if file require access token before client can read it
  - **LongFileNameSubsectionSize [4 bytes]:** Size of the long file name subsection.
  - **LongFileName subsection [X bytes]:** Subsection containing long file names in null terminating string concatenated together.

d. Icon files

  - **IconFileData [X bytes]:** Content of an icon file.

Format of the eStream Set



v 0.1

Open Issues

- Where is the metadata associated with the Root directory located? Currently, root metadata is located in the root version table. All other files and directory metadata can be found in their parent directory.

Exhibit D4

eStream 1.0 High Level Design

Version 0.3
31 July 2000

Notes

The following is roughly what's changed since the last version (0.2):

- The functional requirements and use cases have been removed. These will be documented in the eStream Requirements Document in future revs.
- The entire accounting hierarchy (what is a user and account, how are they grouped, at what level does billing take place) is undergoing revision, and has been removed from here for this version.
- Component descriptions should be more consistent now.
- The database of user and subscription information in the client block diagram has been removed. See the notes below.

Known issues

- The mechanism for how a pathname on a client machine translates into a globally unique FileID for any eStream server is unclear. This is a major design issue that crosses many components on both the client and the servers.
- The accounting hierarchy and its impact on this design are missing.
- If and how copy-on-write will work for writes to the Z file system is quite unknown.
- Which server manages user/account/group/subscription data is quite uncertain. Representing this by a data cylinder was wrong, and I removed this. However, all the interfaces specified below for an "ASP web server" are now just plain wrong, and the server team needs to suggest the appropriate changes to the HLD for the server topology.
- The "Server Data Objects" section at the end of this document needs to be rewritten, in terms of interfaces that client and server components supply to support these data.

Introduction

This document describes the high level design for the eStream 1.0 product. The organization used is:

- Definitions
- Block diagrams for both the client and server portions, showing all major components
- Each component, generally broken down by
    - purpose

- functionality
- global data managed, if any
- interfaces for use by other components

To understand the problem being solved in this design, see the "eStream Requirements Document" for information.

Definitions

account

A billing entity consisting of a set of users and subscriptions

user

An entity authorized to use an account

subscription

An agreement between user and the ASP to use an application under terms of licensing.

license

Legal right to use an application at any given time.

account admin

A special kind of user who can add/delete other users from an account.

server admin

Administrator for all the eStream servers and database.

AppID

A unique representation of an application. There is one to one mapping of AppIDs to apps.

FileID

Within an application, a unique representation of a particular file.

access token

This represents the right to run an eStreamed application. The client must acquire an access token before accessing any file (e.g., executing) in an eStreamed app.

application

An *application* is the set of all files and directories, served by an eStream server, that make up a subscribed application. For example, any executable file, DLL, icon file, or data file associated with an eStreamed version of FrameMaker is part of this application.

application installation

This is the process of locally installing all bits necessary to execute an application via eStream. Most files in the application can be read or executed via the eStream file system; some must be installed locally. Some configuration data must also be downloaded and processed to allow seamless execution of apps.

app install block

This is what needs to be downloaded and installed during application installation. It might consist of:

- all configuration files that must be installed on the client machine
- all registry spoofing information required to run the app
- all file spoofing information required to run the app
- the names of all files and directories that make up the application
- initial prefetch data
- initial pages for critical application files

It's quite possible that this app install block is actually an executable file or a DLL that performs all actions to make an application ready to run, rather than simply a block of data.

ASP ID block

An *ASP ID block* consists of all the information about the applications available to a given user for a given ASP, on a given client machine. Since a user might belong to multiple accounts for an ASP, this represents all subscribed applications for all accounts for that user.

Such data might consist of:

- user name
- password
- ASP contact info (IP address, URL, etc.)
- list of subscribed apps
    - is the app installed on this machine?
    - serial number for app
    - ADRM server(s) to use for validation of this app
    - App server(s) to use to retrieve the app install block

  - App server(s) to use to retrieve app file data
- last time stamp when the ASP was checked for new subscriptions

client certificate

The *client certificate* for a client machine is a digital signature used to identify it to eStream servers. We anticipate this to be used for requests that don't require an access token -- i.e., a valid license. For example, retrieving the app install block, or data for eStream application files that don't require license validation.

client machine

This is a computer on which an eStreamed application executes. It may host multiple registered users, and a single user can install the eStream client on multiple client machines.

eStream client

This is the aggregate of all the software required on a client machine to subscribe to, install, and execute an eStreamed application.

eStream file system

The *eStream file system* (or EFS) is a distributed file system with prefetch and caching functionality. All file data and metadata accessed through the EFS is subject to license validation before being available from a server.

license validation

The act of validating a license means gaining an access token. Generally, before an eStream application can be run on a client machine, the validity of using this application by the current user must be checked. This check is done when certain files associated with the application are accessed; an eStream server is contacted to perform this check and return an access token.

subscription serial number

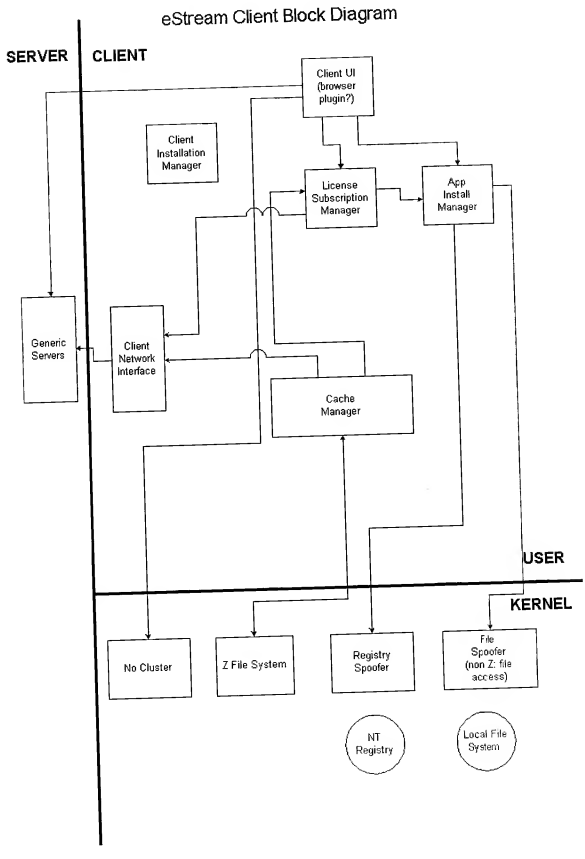
Each application that a user is associated with a serial number. This identifies *both* the application and the user uniquely, and hence can be checked easily during license validation.

Block diagram

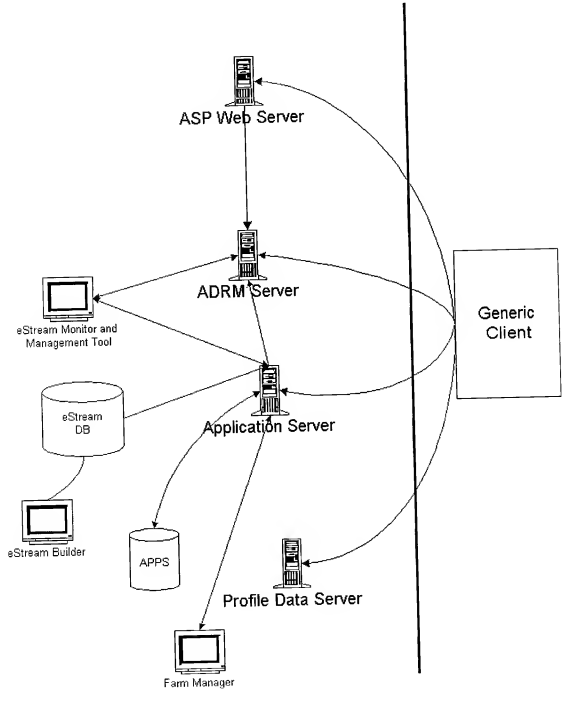
The following are simple block diagrams of the client and server components. Some conventions:

- A box represents a **logical eStream component**. A component may exist as a distinct process, or it may be grouped with other components into a common process.
- A line between components represents an interface call from one to another. If A calls B, there's a arrow on the end of the line at B. If A and B call each other, there's an arrow on both ends of the line.

Note that data stores are **not** represented in these diagrams; if a data store is centrally managed, then there is a component that has interfaces to allow access to these data.


eStream Client Block Diagram
SERVER
CLIENT
Client UI (browser plugin?)
Client Installation Manager
License Subscription Manager
App Install Manager
Generic Servers
Client Network Interface
Cache Manager
USER
KERNEL
No Cluster
Z File System
Registry Spoofer
File Spoofer (non Z: file access)
NT Registry
Local File System

eStream Server Block Diagram



Component descriptions

Client components

The client components are all identified in the block diagram above. Very briefly, some points:

1. A web browser on the client machine will be used for most user interface requests: subscribing to applications, requesting subscription and payment information, and so forth. Configuration of the eStream client software will be done using a UI which may be different from a web browser. Some thoughts on this are listed below.
2. The eStream cache manager is the heart of the client software, and is the component that actually requests file data from the servers.
3. The license subscription manager has the task of tracking all valid subscriptions to applications from an ASP, and tracking which applications have, or need, a license validation to access files.
4. The app install manager's task is to wait until it's told to install a newly subscribed application, and then do so. It also keeps track of what needs to occur when uninstalling an application.
5. The client network interface simply takes requests from the rest of the client components, and forwards them on to the appropriate eStream servers.
6. The eStream file system (EFS, aka the "Z" file system) is a standard kernel-mode network redirector. It presents the normal FS interface to the rest of the NT executive, and requests data from the eStream cache manager to satisfy requests made of it.
7. The registry and file spoofers are kernel-mode drivers that monitor registry calls and file open requests, respectively.
8. The No Cluster component is a very simple kernel-mode driver that disables page clustering for reads.

Installation Manager

Purpose

Installation of the eStream client software is not different than installing any other client software package such as Winzip or Office. The eStream client installation is separate from the installation and configuration of eStream subscribed applications. Some of the possible pieces that eStream would need to be installed are listed below.

1. Device Drivers
2. Applications Executables
3. Application Components
4. Shared Components
5. Registry entries

6. Shortcuts and Start menus
7. Help Files
8. Uninstaller

Once the pieces of the application to be installed are brought together then an install program must be constructed.

Functionality

The Installation Manager consists of the following sub-components

Installshield

Installshield is the industry standard for building installation sets for Microsoft Windows. Installshield will take a set of executables and data files and create a media installation. The Installshield environment provides a scripting language that will allow a high degree of customization of target installation. The essentials issues for any installation are.

1. How much of the application does the user wish to install?
2. Is the users system capable of running the application?
3. Where does the user wish to install the application?
4. Does the user have enough space to install the applications?

Installshield has a wizard that will set up a project. When the install shield program is compiled a media must be specified. The most common media types are floppy, CD Rom, and Web media builds. For eStream we may have to ask the clients to reboot the machine since we are installing kernel mode components that might need a reboot to take effect.

Install From the Web

This program is another product that is sold by Installshield that will take a complete installation set and create a single executable .exe that can be easily downloaded from a web site.

Uninstaller

Installshield will provide an uninstaller when it builds the install program.

Registry Settings

There are three ways that Installshield can patch the system registry.

1. Run regsvr32.exe on self-registering .dll files. When the uninstaller is run it will use regsvr32.exe /u to un-register the .dll file.
2. Patch the registry statically.

3. Patch the registry based on Installation Options from the install shield script program.

Artwork

The Installshield program for eStream will require a splash screen and possibly one or two other artwork components.

eStream Client UI Module

The eStream Client UI module is a client component, currently expected to be running in user space.

Functionality

The eStream Client UI module supports reporting eStream-specific error & informational messages to the client user and soliciting replies when appropriate. It allows the eStream client user to view and change the list of applications currently installed on the client system and the list of ASP accounts currently known to the client system.

Interfaces

ReportMessageToEStreamClientUser(IN message)

Display specified message in EStream Client UI message window.

QueryEStreamClientUser(IN message, OUT response)

Display specified message in EStream Client UI message window and solicit yes/no response for return to caller.

Installed Applications UI

The Client UI interface allows the user to request that the list of the applications currently installed on the client be displayed. The Client UI Module gets this list by calling AIM/GetAppInstallList().

The Client UI interface allows the user to select an application from this list to be uninstalled. The Client UI Module calls AIM/UninstallApp() to accomplish this.

The Client UI interface allows the user to enter the information necessary to get a new application installed. The Client UI Module calls AIM/InstallApp() to accomplish this.

The Client UI interface allows the user to request that the list of applications currently installed on this client be exported to a file, in a form which would allow that list to be

imported on another client. The Client UI Module calls AIM/ExportInstalledApps() to accomplish this.

The Client UI interface allows the user to request that a list of applications that was exported by eStream running on another client be imported from a file & installed. The Client UI Module calls AIM/ImportAndInstallApps() to accomplish this.

Known ASPs UI

The Client UI interface allows the user to request that the list of ASPs currently known to the client be displayed. The Client UI Module gets this list by calling ???.

The Client UI interface allows the user to select and connect to an ASP in the list. The Client UI Module accomplishes this by ???.

The Client UI interface allows the user to select an ASP from this list to be deleted. The Client UI Module calls ??? to accomplish this.

The Client UI interface allows the user to enter the information necessary to record information about a new ASP account. The Client UI Module calls ??? to accomplish this.

The Client UI interface allows the user to request that the list of ASPs currently known to this client be exported to a file, in a form which would allow that list to be imported on another client. The Client UI Module calls ??? to accomplish this.

The Client UI interface allows the user to request that a list of ASPs that was exported by eStream running on another client be imported from a file & installed. The Client UI Module calls ??? to accomplish this.

eStream Cache Manager

Purpose

The eStream Cache Manager (ECM) is a client component, currently expected to be running in user space. Its goal is to:

- Handle all file requests from the eStream file system, either by using previously cached contents or requesting the contents from a server.
- Intelligently use prefetching of file data to reduce latency of pages requested from the EFS.
- Work with the license subscription manager to insure that all applications have appropriately validated licenses before their files are accessed.

Functionality

The ECM handles the volatile & non-volatile eStream cache on the client machine. It performs demand fetching and prefetching from the appropriate server(s), using profiling data or heuristics. Based on the client's observed behavior, it compiles updated profiling data, which may periodically be uploaded to a server.

Interfaces

The ECM takes requests from the EFS driver, and makes requests to the client network and LSM modules.

In the descriptions below practically every call could fail for a variety of reasons. The associated error handling paths are not shown at this level of the design.

Open/Create(IN Filename, IN FileOptions, OUT Handle)

Called from the EFS.

This does the following basic tasks:

- If the filename and mode options correspond to a FileID that is already known and legal to use (from it's cache), it can just return the handle for this file.
- Otherwise, it must ask the LSM for an access token for this file. (This request may simply return an access token previously created for other files making up the application.)
- Launch any prefetching and/or active cache loading activities desired for the new app.
- Request a file handle from an appropriate app server, via the client network component.
- Return the handle to the caller

Close(IN Handle)

Called from the EFS.

This basically:

- Informs the LSM that the file is being closed
- Unloads (or marks as victims) app's cached entries as desired

Read(IN Handle, IN ReadOffset, IN ReadLength, IN BufferPtr, OUT BytesRead)

Called from the EFS.

This does the following:

- Update the profile data for this file
- Check the cache for the requested data
- If not there, request the appropriate pages from an app server, along with any page prefetches that are needed, and place the retrieved data in the cache
- Fill in the buffer and return the number of bytes written to this buffer

Write(IN Handle, IN Buffer, IN Offset, IN Length, OUT BytesWritten)

Called from the EFS.

This will use some copy-on-write scheme. It may be as simple as locking the written structures in the cache.

Global Data

ActiveAppsData Structure

Table of data with an entry for each application that is currently active on the client. Fields for each entry are listed below.

- AppPrefix
- AccessToken
- ServerName
- FilesOpen
- LocalPathName

FileID Type

A globally unique identifier defined for each file associated with an eStream application. All EStream-managed files have these identifiers to allow a common & unambiguous method of file referencing between clients & servers & to simplify switching the client to an alternative server.

ProfileData Structure

Exact contents of this data structure will be defined in the low level design phase; at this point, assume predecessor/successor pairs w/counts.

Volatile & Non-volatile Caching Structures

Exact contents of these data structures will be defined in the low level design phase.

License Subscription Manager (LSM)

Purpose

The LSM tracks current subscription information and determines the need for license validation. It is informed of subscription changes from the client UI, and is queried by the ECM to validate accessibility to different applications, based on the license model for the subscription to that application.

Functionality

The LSM tracks the users subscriptions to different ASPs; it is part of the client component downloaded on a client machine. The LSM starts running when the client component starts running, and is remains active until it stops.

The LSM has a few major tasks:

1. Keep track of what subscriptions the current user has available from all ASPs
2. Determine which application a given file is a part of
3. Acquire an access token to validate a license for file requests that require one

There are two ways that the LSM updates its list of known subscribed applications:

1. It may be informed of new subscriptions, or of applications that are unsubscribed, by the client UI, as part of a browser plugin in conjunction with an ASPs web site.
2. It may asynchronously poll an ASPs ADRM servers to get updated lists of subscribed apps.

When the users start running any of the subscribed eStream applications—i.e., when any eStream'ed file is opened—the ECM queries the LSM before servicing any requests. The LSM checks to see which subscribed application this file belongs to, and, if necessary, gets the appropriate access tokens from ADRM servers along with the identities of application servers that can be used to run the applications; it uses the client certificate obtained when the connection to the ASP was made. At the same time, the LSM can decide to cache the access tokens and the identities of the application servers and decide to serve them directly from its cache.

The ECM informs the LSM when files open and close, and determines from this when applications start and end. The LSM keeps track of when access tokens are expiring and can request for additional access tokens when applications are running and the current one is about to expire.

Global Data

The global data managed by the LSM includes

1. The ASP ID Blocks which are obtained when the user on the machine establishes a connection with an ASP from which the user has subscribed applications.
2. The access tokens and the identities of the applications servers that are obtained from the ADRM servers when the user tries to run the applications.

Interfaces

The LSM exposes the following set of APIs to the client UI:

SubscribeApp(IN ASPId, IN AppID, IN LicenseInfo)

This routine in turn will call the App Install Mgr to install the application on the client machine. This will return a Boolean stating success or failure.

UnsubscribeApp(IN ASPId, IN AppID)

This routine will NOT implicitly uninstall the application. Applications must be explicitly uninstalled. This will return a Boolean stating success or failure.

GetAppList(OUT SubscribedAppList)

This routine will return a pointer to a list of subscribed applications on the client machine.

The LSM exposes the following set of APIs to the ECM:

CheckAccess(IN Path, OUT Root)

The LSM establishes a correlation between the Path and the AppID by querying the App Install Mgr. This routine in turn may contact the ADRM server for appropriate access tokens. This will return a Boolean stating success or failure. At the same time Root will get set to the head of the path that identifies the application so that the file system can use the same access token for everything under "Root".

BeginApp(IN AppID)

To indicate the start of an application. **Note:** this may happen implicitly during CheckAccess().

EndApp(IN AppID)

To indicate the end of the application. **Note:** this may happen implicitly during CheckAccess().

The LSM makes the following API calls.

1. InstallApp(ASPId, AppID) to the App Install Mgr to install the subscribed applications.
2. GetAppId(Path, &Root) to the App Install Mgr to get the AppId from the Path. "Root" is explained above.

The LSM sends messages to the ADRM server for getting access tokens. When a user goes to a new machine and installs the eStream client, the LSM obtains the subscription information from this server when the user first establishes a connection with it.

Application Install Manager (AIM)

Purpose

The AIM is the contact point for installation and uninstallation of applications on a client machine. It gets the requests from the LSM to install applications when the user subscribes to applications, and it gets requests from the Client UI to uninstall applications.

Functionality

The AIM manages application installs on the client machine. It keeps track of what applications have been installed on the client machines, where they have been installed and the various components that are part of the installation. It contacts the application servers (obtained from the ASP ID block) to get the AppInstallBlock. This may be a data block, an application or a dll. The AIM uses the AppInstallBlock to then make the appropriate calls to the Registry and File spoofers; to install some files on the local disk; to "warm" the cache and to update the start menu and other short cuts as needed.

Global Data

The Global Data managed by the AIM includes –

1. The AppInstallBlock obtained from the app server that is used to do the installation.
2. The AppID->Path co-relation that is required to check for access privileges.

Interfaces

The AIM exposes the following interfaces –

InstallApp(IN ASPId, IN AppID)

To install the application using a specific ASP server to get the AppInstallBlock.

UninstallApp(IN AppID)

To uninstall the application from the client machine.

GetAppId(IN Path, OUT Root)

To return the AppID given the Path that is being used to open a file/directory on the eStream file system.

GetAppInstallList(OUT InstalledAppList)

To get a list of the applications currently installed.

The AIM makes calls to the registry and the file spoofers using the AddRegSpoofEntry, AddFileSpoofEntry, etc. APIs.

eStream client network component

This section deals with the components that communicate with the servers.

Purpose

The client network component is the common point of connection between the rest of the eStream client components and the various eStream servers. Any client module that calls an interface of a server does so through the network component.

This component is basically stupid. It knows the protocols needed for communicating with the various servers, and it can encode the requested messages via these protocols, but it doesn't try to be smart with regard to failover, or authentication rejection, or other error conditions. The network component lets its caller deal with such matters.

One design assumption here is that data is received from an eStream server only in response to a request it has made of this server. In other words, all requests originate with the client, never from the server.

Functionality

The client network component communicates with the following servers for the types of requests listed.

ADRM server

1. Validate a user for this ASP and get subscription information
2. Validate a license for a subscribed app

App server

1. Open a file/directory for a subscribed app

2. Various file requests on a previously opened file/directory

Global Data

Probably none (?).

Interfaces

ValidateUser(IN ADRMServer, IN AspAndUserData, OUT SubscriptionInfo)

This interface is called by the LSM; it is used both to validate a user and get updated subscription information for a given ASP.

ValidateLicense(IN ADRMServer, IN APPId, IN ClientCertificate, OUT AccessToken, OUT AppServerList)

This is called by the LSM, to get an access token for an application before its file can be accessed.

AppOpenFile(IN AppServer, IN AccessToken, IN FileDesignator, OUT Handle)

This is called by the ECM, to for any eStream file. Note that this is also used to retrieve an AppInstallBlock, when requested via the AIM. **Note:** the FileDesignator is still undergoing design.

AppReadFile(IN AppServer, IN AccessToken, IN Handle, IN OUT Buffer, IN Offset, IN Length, OUT BytesRead)

This is called by the ECM.

UploadAppProfileDataRequest(IN ADRMServer, IN ProfileData, OUT Success)

It's unclear who calls this!

eStream File System Driver

The file system driver interfaces with the operating system's installable file system facilities, forwards file system requests that it cannot directly satisfy to the ECM, and uses the NT File Cache to optimize repeated accesses to the same data. This component will be very operating system specific, while the interfaces it exposes to the cache manager will be (mostly) OS independent. The file system driver resides in kernel space and implements a portion of the entire eStream file system. Other components, such as the cache manager, the client network interface, and the app servers, implement the rest of the eStream file system. These other components are not necessarily kernel-mode resident.

Functionality

The eStream file system driver (EFS) will send most requests from the operating system to the cache manager. It will interface with the standard NT File Cache Manager to avoid sending redundant requests to the cache manager. And it must support functionality for the ECM to notify it when the data structures it has cached have become invalid.

Global Data

The only globally-visible data managed by the file system driver are various things that it may cache. This includes both file data pages as well as directory contents. These data are relevant to the ECM, because it may want to invalidate the contents of the caches if it finds a newer version of a data page or finds that (visible) directory contents have changed.

Interfaces

These are the logical interfaces that are exposed to the ECM. The EFS has standard file system interfaces that are used by the NT Executive, but these are not listed here.

InvalidatePage(IN FileHandle, IN PageOffset)

Invalidates the specified page for the specified file handle in the cache.

InvalidateDirectory(IN DirHandle)

Invalidates the specified directory's contents in the cache. This may result in the eStream file system driver sending directory change notifications.

ShutdownFileSystem(IN Force)

Attempts to shut down the file system. If Force is true, the file system will be shut down regardless of whether any processes still have handles that are open on the file system. If Force is false, this routine will return an error if there are any open file handles. After the file system is shut down, any attempt to access the file system will result in errors rather than being forwarded on to the cache manager, until StartFileSystem is called.

StartFileSystem()

Causes the eStream file system to begin accepting requests and forwarding them to the eStream cache manager.

Virtual Memory Clustering Disabling Driver

Purpose

The VM clustering disabling driver (aka NoCluster) disables virtual memory clustering under Windows. While we don't fully understand all the implications, using this driver substantially reduces the average file system paging request size and can dramatically improve performance of eStream, especially on slower connections.

Virtual memory clustering, as implemented in Windows NT/2000, is intended to improve performance when paging to and from physical disks. If possible, we would like to disable clustering only for those threads/processes that will be doing a significant amount of I/O to the eStream file system.

Functionality

The VM clustering disabling driver maintains a set of criteria for threads whose clustering should be disabled. It will make decisions about which threads should have clustering disabled without contacting any other components.

Global Data

The only data managed by this component are the criteria for selecting threads whose clustering should be disabled. In the simplest implementation, this would be nothing, and the driver would disable clustering for all threads. Whatever tables it uses will be resident in the kernel, and the driver will be able to access them as needed without making calls to a user-mode component to read them.

Interfaces

The interfaces for this component are minimal. Clustering may be enabled or disabled, and the criteria for which threads to manipulate can be changed.

StartDisablingClustering()

This interface notifies the driver that it should begin disable virtual memory clustering, using the currently specified criteria.

StopDisablingClustering()

This interface notifies the driver that it should stop disabling clustering. Note that due to implementation problems, we do not support actual unloading of the driver, though it can be removed from the system by a reboot.

ChangeClusteringCriteria(IN Criteria)

This interface allows the caller to cause the driver to change the criteria it uses to select threads it should manipulate. The criteria might be specified in the interface, or it might specify a file that the driver should read for the new criteria.

QueryClusteringCriteria(OUT Criteria, OUT Active)

This interface allows the caller to find out what criteria the VM clustering disabling driver is currently using, and whether or not it is currently active.

File Spoofer

Purpose

The purpose of the file spoofer is to redirect file system accesses from some non-eStream drive. This may be necessary in order to support applications running under eStream that are hard-wired to access files in a specific location. The file spoofer may also be used if we are interested in providing a version of some system file different from the one actually on the client machine.

Functionality

The file spoofer will intercept File Create calls for files that we are interested in spoofing and ensure that these creates are redirected to a file we specify. The redirection could be to a file on the Z file system, or to another, non-eStream'ed file.

File open is a very common occurrence, so the file spoofer must operate quickly. The file spoofer should maintain in-kernel whatever data structures it needs to make a spoofing decision.

Global Data

The file spoofer must maintain a database indicating which files to spoor, which file to replace them with, and possibly which processes should be spoofed. All of this information must be kept in-kernel so spoofing decisions can be made quickly. This database may change depending on which eStream apps are currently installed or running.

Interfaces

StartFileSpoofing()

Causes the file spoofer to begin file spoofing, using the current spoof database.

StopFileSpoofing()

Causes the file spoofer to stop spoofing, but does not change the spoof database.

AddFileSpoofEntry(IN SpoofEntry)

Adds an entry to the spoof database. It is not necessary to stop and restart the spoof database to add an entry.

RemoveFileSpoofEntry(IN SpoofEntry)

Removes an entry from the spoof database. It is not necessary to stop and restart the spoof database to remove an entry.

QueryFileSpoofDatabase(OUT SpoofEntryList)

Queries the current contents of the spoof database.

ReplaceFileSpoofDatabase(IN SpoofDB)

Replaces the entire spoof database. It is not necessary to stop and restart the spoof database to perform this action, and it is considered atomic.

1.4 Registry Spoofer

Purpose

The purpose of the registry spoofer is to provide to eStreamed and other applications registry entries for eStreamed apps, and to capture registry writes by eStreamed apps so they can be purged from the registry or shipped to other clients for configuration ubiquity.

Functionality

The registry spoofer must redirect registry reads and writes. Because registry accesses are quite common, the redirector should be able to service registry spoofs without forwarding the requests to a user-level process.

Global Data

The registry spoofer maintains a spoof database of which registry entries to spoof, and which processes to spoof them for. This database should be kept in-kernel so that the spoof decisions can be made quickly. The spoof database may change depending on what eStream applications are currently installed or running.

Interfaces

StartRegSpoofing()

Causes the registry spoofer to begin spoofing using the current database.

StopRegSpoofing()

Causes the registry spoofer to stop spoofing. This does not change the registry spoof database.

AddRegSpoofEntry(IN SpoofEntry)

Adds an entry to the registry spoofing database. It is not necessary to stop spoofing in order to do this.

RemoveRegSpoofEntry(IN SpoofEntry)

Removes an entry from the registry spoofing database. It is not necessary to stop spoofing in order to do this.

ReplaceRegSpoofDatabase(IN SpoofDB)

Replaces the entire registry spoofing database. It is not necessary to stop spoofing in order to do this, and it is considered an atomic operation.

QueryRegSpoofDatabase(OUT SpoofEntryList)

Queries the contents of the registry spoof database.

Server components

The servers described below are **logical servers**. Note that a single server machine can serve all functions for a small ASP; alternatively, farms of servers can be used to provide the functionality of a single logical server.

NOTE: The distribution of servers and the functionality provided by them are somewhat uncertain to date. In particular, exactly who manages the actual accounting, user, and group data is undecided. The group producing the LLD for the eStream servers need to flesh this out. For now, this document assumes that the ADRM server ultimately manages these data, and supplies interfaces to callers to access these data.

The servers described are:

1. An ADRM server handles user/account/subscription data management, as well as validating licenses.
2. An ASP web server is a front-end for requests to add users, subscribe to applications, and do various user and application level queries. Generally this forwards these requests to an ADRM server.
3. An application server handles requests to open and read eStream files.
4. A profile data server will receive uploaded profile data from a client machine to enable better initial profile and prefetch maps in eStream sets.

ADRM Server

Purpose

The Account/Digital Rights Management (ADRM) server is responsible for:

- Managing data related to users, the groups they belong to, and the applications they are subscribed to
- Validating the licenses for applications executing on clients
- Tracking all outstanding licenses currently in use

Functionality

Client machines sends requests to the ADRM server to add or delete subscriptions, to receive an access token to execute an application, and to manage their account/group/user relationship.

Access tokens have an expiration time, so the client must reacquire them at regular intervals. When an eStreamed application exits, the client informs the ADRM server to release the access token. Any outstanding access token not released or reacquired within the expiration time will be automatically released by the server.

Interfaces

AcquireAccessToken(IN UserInformation, IN AppId, OUT AccessToken, OUT AppServerList)

This is called by the eStream client to gain validate a license before executing an application.

This is used to insure that a user has the right to use a particular app in a subscription from a specific account. The server returns an access token and a list of app servers from which the client can access the application file data. If the user doesn't have a valid license to use the requested application, a failure message is sent to the client. The server writes the start time of this application usage into the database for billing processing.

RenewAccessToken(IN OldAccessToken, OUT NewAccessToken, OUT AppServerList))

Note: This may just be replaced by AcquireAccessToken()

This is called by the eStream client.

The server receives a message from a client to renew its access token before the expiration of the token. The server returns a new access token and a list of app servers. This allows the server to redirect the client to a different app server in case it knows of changes to the list of available servers. Once the token is expired, the ADRM server

writes the end time of this usage information into the database and the client must reacquire the access token before files for this application are available to it.

ReleaseAccessToken(IN AccessToken)

This is called by the eStream client.

The client returns the token to the server when the eStream app terminates so other clients can acquire the token. The server writes the end time of this usage information into the database for billing processing.

AddApplicationServer(IN AppServer, IN ApplicationList)

This is called by an application server.

The ADRM server is informed of the availability of a new application server. The ADRM server adds this new app server to its list of app servers.

RemoveApplicationServer(IN AppServer)

This is called by an application server.

The ADRM server is told of the removal of an app server. It must remove this app server from its list of such servers to prevent any clients from using that server.

AddApplicationServerApplications(IN AppServer, IN ApplicationList)

This is called by an application server.

The ADRM server is informed of the availability of a new application on a given app server. The ADRM server adds this new app to the list of applications that the server has available.

RemoveApplicationServerApplications(IN AppServer, IN ApplicationList)

This is called by an application server.

The ADRM server is told of the removal of an application for an application server.

GetTrafficHistory(IN TimeCriteria, OUT HistoryData)

This is called by a server UI tool, or possibly by other ADRM servers.

The administrator monitoring, reporting, and management tool UI program can query the ADRM server for load information. The server logs all client requests to acquire access token. This raw information can either be sent directly to the UI program or it can be preprocessed before sending to the UI program.

GetErrors(IN TimeCriteria, IN ErrorType, OUT ErrorList)

This is called by a server UI.

The admin UI program can query the ADRM server for any errors. The errors can be categorized by type of errors or errors that occur between certain time periods. A small sample of the possible ADRM server errors includes: client access token timeout, failure to read user information from the account database, failure to get the license information, failure to write usage information into the database, etc…

GetIllegalAccesses(IN TimeCriteria, IN AccessType, OUT AccessList)

This is called by a server UI.

The admin UI program can query the ADRM server for any illegal accesses. The illegal accesses can be categorized by type and time period. A small sample of the possible ADRM server errors includes: failure attempts to access ADRM server with bad password repeatedly in a small time period, failure attempts to use a particular license, any access attempts from a non-typical IP address ranges for a particular account, etc…

GetAppID(IN AppName, OUT AppID)

This is called by a server UI.

Returns a unique identifier associated a particular application

SetApp(ID IN AppName, IN AppID)

This is called by a server UI.

Stores a unique identifier associated a particular application

Application Server

Purpose

The application server is there to handle read requests for files accessed by eStream clients. Any file accessed on a client through the EFS can have this read request passed to an app server.

Functionality

This will be the hardest working eStream server. It will respond to both synchronous (demand fetching) and asynchronous (prefetching) page requests from many different clients, for many different types of applications and files within those applications.

Interfaces

GetFileInfo(IN AccessToken, IN FileID, OUT FileInfo)

This is called from an eStream client. Given any file within an eStream application, return metadata about it. The access token is provided for validation.

ReadFile(IN AccessToken, IN FileID, IN Length, IN Offset, OUT Buffer, OUT BytesRead)

This interface is called by an eStream client, and will allow the client to access any eStreamed application file and AppInstallBlocks. How the FileID for an AppInstallBlock is achieved is unclear at present.

OpenFile() / GetFileID()

Note: This is a placeholder for an API that may be needed. This depends a lot on the eventual communications between client and server for associating a file pathname with a FileID.

ASP web server

Purpose

This describes, of course, only those interfaces on an ASP web server that relate to handling eStreamed applications.

Logically, the ASP web server is the backend web interface for user requests—e.g., get billing information, subscribe to a new app, or request a list of all possible apps a user can subscribe to. In the current model, the web server doesn't actually handle these requests, but instead passes them on to the appropriate eStream-centric server.

NOTE: The following interfaces are not updated from the previous version! They were written with the assumption that the web server actually manages all the data described above. We need the server team to suggest the changes that should take place here!

Functionality

Interfaces

AddADRMServer()

The ASP Web Server is informed of the availability of a new ADRM server. The ASP Web Server adds this new ADRM Server to its list of online ADRM Servers. Periodically, the ASP Web Server can query the list of ADRM Servers for its load

information. When a new client connects to the ASP Web Server, the client can be informed of the subset of ADRM Servers with the least load.

Callers: ASP Web Server

Input:

- ADRM Server IP
- list of Account Ids that the ADRM Server supports.

Output:

- success or failure

RemoveADRMServer()

The ASP Web Server is told of the removal of an ADRM Server. It must remove the ADRM Server from it's locally cached list of ADRM Servers to prevent any future clients from using that particular ADRM Server.

Caller(s): ASP Web Server

Input:

- ADRM Server

Output:

- success or failure

ValidateSubscribedUser()

Inputs:

- SubscriptionToken

Outputs:

- success/failure

Validate subscribed user is called by the ADRM server to check out a license.

- Find account(accountNumber)
- Find user(username)
- Find license(SubscriptionID)

If a license is found to be available, check it out and return OK else return NO_LICENSE_AVAILABLE.

Add/RemoveAccount()

Input:

- ownerUsername
- ownerPasswd
- billingInfo

Output:

- accountNumber

Creator must supply info for the first user, who is also the account owner.

Add/RemoveSubscribableApp()

Input:

- application
- AppServer locations
- name description

AddAccountUser()

Input:

- account number
- user name
- initial password

Add/Remove/Increment/DecrementFloatingLicense()

Input:

- account number
- list of users
- subscription
- number available

Add/RemovePerUserLicense()

Input:

- account number
- user
- subscription

CompileAccountUsage()

Input:

- account number

Add up and report all the usage by members of the account.

ClearAccountUsage()

FreeLicense()

Input:

- user
- subscription

Frees a license that had been previously checked-out by a user.

GetUserPriveleges()

Input:

- account number
- username

Check privileges of a logged in user for the purposes of allowing user/subscription management and other account changes

ShowCheckedoutLicenses()

GetAccountInfo()

Input:

- account number
- username

ListPossibleSubscriptions()

Input: none

ListCurrentSubscriptions()

Input:

- accountNumber

CreateSubscription()

Input:

- account number
- license data (depends on license type)
- ApplicationPackage

Output:

- subscriptionID
- ADRM server names.

Builder

Does not talk to any other module. Probably an associated set of tools managed via a script plus manual procedures that create intermediate data files, and finally produce the eStream set.

Farm Manager

Simply takes user commands and input, activating the following functions on the actual Application Server process:

StopServer(boolean graceful)

ConfigureServer(config_parameters)

EnableEstreamSet(appID) - informs the app server that the set is ready to be served

DisableEstreamSet(appID) - opposite of above

Functionality required for upgraded Estream sets may extend this, management of farm-level stuff would be an extension of the above. Synchronizing the availability of apps between the Application Server and ADRM/Account DB must also be handled; at least initially, a human administrator should be able to flips the final switch.

Monitor

The monitor utility is responsible for monitoring the overall health of the system. It is responsible to report server status, server traffic, illegal access etc. It will ping the Application Server and the ADRM server to gather the statistics and display them.

Server data objects

This section needs work! What should be here?

eStream Set

What is an eStream set? It consists of:

- Page prediction map, indicates likelihood a page will be referenced successively after another page. Used to enable accurate prefetching by the client.
- Spoofer info, stuff to initialize register & file spoofer to enable application to run on the client. (ITARD & "Spoofed file mapping list")
- Application content. This includes all files of the application in possibly multiple subtrees (stuff from C:, Z:, Common Files, etc.). These are all then placed under a single application root directory which is "mounted" under the eStream file system under the application's directory ("Microsoft Office" or whatever). This directory structure is then processed to create a special eStream metadata file to represent it, and map file names to FileIDs, one assigned to each file for the app, used by the client. All of the application files can then be placed into a single large file with a FileID index in front to allow the **Application Server** to map requested FileIDs to offsets in the large application content image file.
- The above takes care of everything in the AppInstallBlock except for possible COM dlls that might be necessary (TBD what mechanism generates these).

Account/Subscription Database:

Description:

The Account Subscription database manages all the data required to manage accounts, applications and subscriptions for an applications service provider. It is used to verify users and subscription rights, to log usage, and compile billing for application rental. This document describes the data model for the db, the list of accessor APIs, and finally, a list of scenarios that exercise these APIs.

Data Model

The following is a description of the data model for the Account/Subscription Database.

There are two types of records described.

Static Objects are objects that are stored persistently in the Account/Subscription database. The attributes of each record are divided into two categories:

- Owns: specifies data that is actually contained in and managed by the object. These are the attributes that make the object what it is. For example: an account is not an account unless it has billing information. Each item (except the ASP) must have only one owner, but may have many references.
- References: Attributes that create associations to other first-class objects, making navigation from one item to another simpler.

Transient Objects represent data structures that are created in order to pass information from place to another. For the most part, they provide shorthand identifiers for static objects.

Each attribute listed below has the format:

Type: name – (optional) description

For attributes that don't yet have a defined type, the type is *undefined*

Static Data (Database records)

ASP – The top-level container.

Owns:

- List<Account>: accounts - all of the accounts
- List<Subscription>: subscriptions - all of the available subscriptions
- List<Applications>: applications - all of the applications currently served by Application Servers

References:

- None

Account - Collection of attributes that make up a single account:

Owns:

- Number: accountNumber – number that identifies this account
- List<User>: users – all of the account users
- Undefined: billingInfo – Currently undefined type

- List<License>: licenses - all of the licenses (per-user and floating) that are managed by the account
- List<UsageRecord> usage – list of records that describe usage of all account users

References:

- User: owner - a user who created the account – must be one of the users

User – a single user of an Account

Owns:

- String: username
- String: password
- Undefined: Role – specifies permissions on the account, i.e. owner, administrator vs. regular user
- Undefined: UserInfo – Real name, contact info etc

References:

- List<License>: licenses currently held by user
- List<License>: licenses - licenses to which the user has access. These might be either per-user or floating licenses, or a combination of the two. This list might also incorporate a means of specifying a preference – for example if a floating license and a per-user license are available, use the per-user

Application – single application that has been made available on one or more application servers.

Owns

- String: name
- String: description
- Undefined: AppServer location(s) – A location may be a host name that must be resolved by the appserver farm, or may be an IP address.
- Number: AppID
- A list of FileIDs for this app.

References:

- None

Subscription – An application or group of applications that have been made available for rental by a user.

Owns:

- Number: SubscriptionID
- String: name
- String: description

References:

- User: user
- List<Application>: applications - application(s) associated with this subscription

ApplicationPackage – *application(s) that can be rented*

Owns:

- Undefined pricing

References:

- List<Application> applications

License – base for other licenses – All licenses support the same APIs – check out, check in

Owns:

- None

References:

- Subscription: subscription – subscription for which this license grants rights

FloatingLicense – one of a fixed number of licenses distributed to a list of users on a first-come first-serve basis. Contains all attributes in License plus:

Owns:

- Number: numTotal
- Number: numInUse

References:

- List<User> holders – the current holders of this license (length = numInUse)
- List<User> allowedUsers – list of users allowed to check out this license

***PerUserLicense** – license tied to a particular user – one desktop (machine) at a time. Contains attributes in license plus:*

Owns:

- Boolean: isInUse
- Undefined – desktopID

References:

- User: allowedUser – the (only) user allowed to pull this license.

UsageRecord – Describes a billable use of the application

Owns:

- Undefined: Start time
- Undefined: End time

References:

- Subscription: subscription
- User: user

Transient Data

AccountNumber – integer that uniquely identifies a single account with an application service provider.

UserName – string that uniquely identifies a single user **within** an account. To uniquely specify a user to an ASP, it is necessary to qualify the UserName with the AccountNumber of which he is a member. All users have

UserVerifier – combination of the AccountNumber, UserName, and UserPassword. Uniquely identifies a user within an ASP

- AccountNumber
- UserName
- UserPassword

SubscriptionID – Integer that uniquely identifies a subscribable application or collection of applications within an ASP.

SubscriptionToken – describes a user-subscription to the ADRM server. Identifies the subscription as well as the user trying to access it.

- UserVerifier
- SubscriptionID

AppID – A unique numeric representation of each eStreamed application. For example, word := 1000, excel := 1001, office := 1002 etc. We should be able to represent software packages using AppIDs.

FileID – Within an eStreamed app, each app-file gets a unique numeric ID.

Exhibit D5

eStream Web Server Load Monitoring Applet Low Level Design

Jae Jung
10/27/00
Modified: 10/30/00

Functionality

One of the requirements for the eStream web server is a facility for monitoring server load (eStream requirements 3.0, 3.2, 3.4, 3.5). Per this document, this facility will be provided by a graphical load-monitoring applet that will be available for deployment at customer sites as part of the eStream web server installation.

The load-monitoring applet will present information in the following formats through a graphical interface:

- Real-time server load information plotted on strip chart
- Historical server load information plotted on line chart
- Multi-server real-time or historical load information on a line chart

Requirements

The following list details the provisional requirements for the load-monitoring applet. The remainder of this design document is based on these requirements.

1. The applet will be able to display server loads in "real-time" as load data is retrieved from the server.

2. The applet will be able to display (in a separate mode from real-time monitoring) historical load information to the extent that this information is available in the database.

3. The applet will be configurable (via applet parameters) for the following settings:
   -- Data retrieval rate, i.e., the frequency with which the applet request new load data from the web application server
   -- Chart window size, i.e., the number of datapoints shown in the chart window at any one time.

4. The applet will be capable of concurrently displaying the load of multiple servers in a clear and concise fashion.

5. The applet will support the displaying of cumulative and average load statistics for multiple servers.

6. In real-time mode, the applet will operate as a strip chart, with the fastest chart speed determined by a global configuration setting in database, typically corresponding to the frequency with which the eStream Monitor inserts load records into the database.

7. The applet will retrieve load information from the database via an http connection to the eStream web app server.

8. The applet will run in browsers with Java 1.0.x and 1.1.x support.

9. Internationalization support. Both the Applet and the backend pieces should be internationalizable.

Description

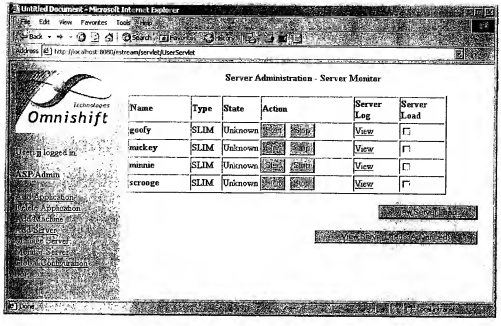
The load monitoring applet is comprised of two components. The first component will manage the retrieval of real-time or historical server load information from the eStream database. The second will consume this data and present this information to the user in a clear and concise graphical format. In addition to the applet, server-side objects will need to be written or extended to service data requests from the applet.

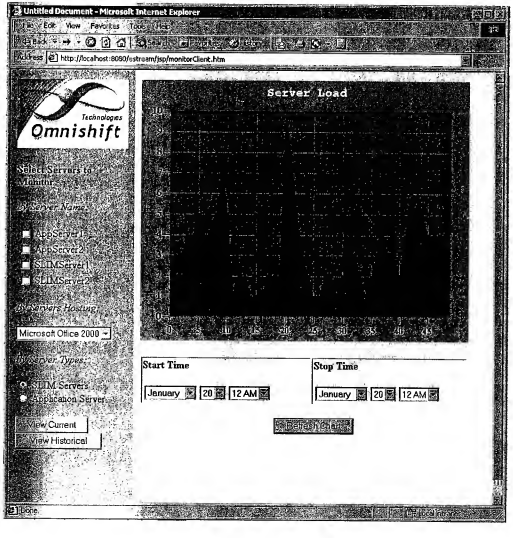
For the alpha-release of the eStream web server, the graphical presentation component will not be written internally; rather a commercially available applet or package will be used. Other aspects of the applet and the server-side components will be implemented to facilitate transitioning to an internally developed graphical component if such a decision is made for later releases.

User Interface Design

The following two screen shots are representative of the way that the load monitoring applet might be used in a particular monitoring/administration Brower interface. The first shot shows a general server administration and monitoring UI and the second shows a detailed load monitoring UI within which the load monitoring applet will be embedded.

The UI options shown in the second shot illustrate some of the reporting options for which the load monitor will be initially configurable. The applet(s) will be readily extensible to generate additional reports and more complex data combinations if desirable.


Untitled Document - Microsoft Internet Explorer
Server Administration - Server Monitor
Omnishift
Name
Type
State
Action
Server Log
Server Load
geofy
SLIM
Unknown
Start
Stop
View
mickey
SLIM
Unknown
Start
Stop
View
minnie
SLIM
Unknown
Start
Stop
View
scrooge
SLIM
Unknown
Start
Stop
View


Untitled Document - Microsoft Internet Explorer
http://localhost:8090/jstream/jsp/monitorClient.htm
Omnishift
Server Load
Start Time
Stop Time
January
20
12 AM
Microsoft Office 2000
SLIM Servers
Application Server
View Current
View Historical

Interface Definitions

The load monitoring applet has two interfaces with other webserver components: <APPLET> tag parameters for its configuration within the web browser page and the HTTP request/response format with the web application server to request and receive load data.

1. **Applet <-> HTML Page Interface (<APPLET> Tag Parameters):**

Parameter	Req'd?	Significance	Default
hostName	Yes	Host name of webserver	None
hostPort	No	Host port of webserver	"80" (int)
chartSpeed	No	Time (in seconds) between chart scroll; also the resolution of the x-axis	"5" (int)
updatePeriod	No	Interval (in seconds) between HTTP requests for server load data. Note if chartSpeed < updatePeriod, the applet buffers load data for presentation. Maximum information latency will at most be equal to this value plus round trip time for the data request.	"10" (int)
chartWidth	No	Width (in ticks) of the applet chart; in conjunction with chartSpeed, implicitly determines the amount of data displayed	"50" (int)
mode	No	"current" for real-time monitoring and "history" for historical	"current"
startDate	No	if mode="history", the starting date for which the historical load chart will be plotted. Note if mode="current", this parameter is ignored.	none; value must be in Java Date format
stopDate	No	if mode="history", the stop date for which the historical load chart will be plotted. Note if mode="current", this parameter is ignored. Note that stopDate override the chartSpeed setting insofar as x-axis resolution is concerned.	none; value must be in Java Date format
numServers	Yes	number of servers to be plotted	1 (int, max 50)
serverName1 serverName2 ... serverName50	Yes	the id of the server(s) being plotted (up to 50 servers can be plotted at once). This **must** correspond to a existing serverID in the Server database table	none
serverData1 serverData2 ... serverData50	No	an initial set of data with which to display the initial chart. Note that these parameter(s) are the default means by which historical load data is displayed. Each set should be a comma delimited list of numbers (float)	none

- Note that applet tag parameters are all strings; where the applet does an internal type conversion, the target type format of the particular parameter has been noted.

- Note that the parameters determining the appearance of the chart (e.g., line colors, grid painting options, custom labels, etc.) are not included in this list. These parameters will either be determined by the parameters available for configuration in the commercial charting applet or TBD at some later date if the charting component is developed internally.

2. Applet <-> Web App Server Interface (HTTP)

Input:

The load monitoring applet will request load data from the webserver by executing the web server's MonitorServlet with the following parameters:

href="/MonitorServlet?action=getLoadData
&serverId=...
&startDate=...
&stopDate=...
&numPoints=...

where:

- serverId is a comma-delimited list of one or more server(s) for which load information is being requested. Note that this serverId corresponds to the serverID attribute in the Server database table.
- startDate is the (exclusive) starting date (in Java Date format) for which the load information is being requested
- stopDate is the (inclusive) end date (in Java Date format) for which the load information is being requested
- numPoints is the number of data points requested between the start and stop dates.

The applet will process the return data points as equally time-spaced points between the requested start and stop dates.

In addition, if startDate=stopDate, only one load data point (i.e., the most recent) will be returned by the servlet.

Output:

The load monitoring applet will expect load information from the web server via HTTP response in the following XML-like format:

<loadData>

```
<serverid=a >
        <LOADLIST>
                <LOAD value=x1/>
                <LOAD value=x2/>
                <LOAD value=x3/>
        ...
                <LOAD value=xN/>
        </LOADLIST>
</server>
<server id=b>
<LOADLIST>
                <LOAD value=y1/>
                <LOAD value=y2/>
                <LOAD value=y3/>
        ...
                <LOAD value=yN/>
        </LOADLIST>
</server>
    ...
</loadData>
```

In the above example, x1 to xN represents load values for the server a (by serverID), and y1 to yN represents load values for server b.

Testing Design

The load monitoring applet and related server-side Java code will need to be tested according to the plan outlined for other web server components in the Web Server/Database Low Level Design (WebServerDB-LLD.doc). Additionally, it should be noted that certain applet-related parameters (i.e., updatePeriod) that affect the frequency with which the applet requests load data from the web server are good candidates for tuning in order that a good balance between UI/measurement response and web server response performance be struck.

Upgrading/Supportability/Deployment Design

Upgrading/Supportability/Deployment Design will follow the model described for the Web Server in the Web Server/Database Low Level; Design (WebServerDB-LLD.doc).

Open Issues

1. How much will it cost to deploy the chosen commercial java applet/package(s) as an OEM installation? We need to find this out before we decide on a commercial

package. Also, another criteria for the choice would be: will we get support. Do we get the source code too?

2. Longer term, where's the cost/benefit breakpoint for the above where it makes more sense to write our own charting applet/package?